Technical Disclosure Commons

Defensive Publications Series

May 22, 2017

Inverse pointer unboxing

Michael Fairhurst

Inverse pointer unboxing

ABSTRACT

Virtual machines (VM) for dynamic programming languages store a combination of 64-bit data types in each 64-bit register. A motivation to store multiple variables in a single register is speed, since access to variables within a register is much faster than access to variables stored within RAM. Current approaches of storing multiple 64-bit values in a single 64-bit register result in undesirable effects such as higher register pressure, increased garbage collection burden, excessive boxing/unboxing steps, etc. Most current approaches cannot distinguish a sufficient number of values in 64 bits, as much is hidden behind 64-bit pointers to RAM. This in turn affects performance of the VM. This disclosure makes use of the floating-point specification to store variables of type double, integer, boolean, etc. in non-canonical pointer space, alongside the pointers themselves. In this manner, more variables are packed in a single register, thereby improving performance of virtual machines.

KEYWORDS

- Pointer tagging
- Virtual machine
- Pointer space
- Boxing and unboxing

BACKGROUND

The memory map of a virtual machine (VM) is classified into regions such as register, heap, stack, etc. Registers are located within the hardware processing unit and offer the fastest speed of access. Stack is a region of memory used for objects that are statically allocated, e.g.,

objects of type *int*, *double*, etc. Heap is a region of memory used for objects that are dynamically allocated, e.g., at run-time. The heap and stack are located within random-access memory (RAM) and are slower to access than registers. Virtual machines (VMs) that use variables mostly located in the registers run faster than VMs that use variables mostly located in the stack or heap. This is because the VM code is executed within the processor, which can access variables stored in registers faster than variables stored in RAM.

This fact is exploited by compilers and by human programmers who try to create VM code with variables that mostly reside in registers, computations that are performed in-place within registers, and computations that require minimum movement of data from register-to-RAM or vice-versa.

Pointer tagging is a technique to maximize storage of variables in registers and reduce data movement between registers and RAM. Pointer tagging exploits two facts:

- The address bus is usually larger than the addressable memory. For example, due to requirements of word alignment, a 64-bit architecture has pointers (which are variables storing address locations) that are 64 bits wide. However, a 64-bit pointer implies a total addressable memory of $2^{64}=1.8\times10^{19}$ bytes, that is, nearly 18 giga-giga bytes. No current computer has such a large amount of RAM. Indeed, by current standards, a computer with 1000 gigabytes (1 terabyte) of RAM is considered extremely rare. Thus, canonical 64-bit pointers utilize only, for example, the lower 47 bits (giving an addressable space of 2^{47}=130 terabytes). The upper 17 bits are copies of bit 47, and are ignored by code.

- A representation of a 64-bit NaN quantity, under floating point arithmetic standards, such as IEEE Standard for Floating-Point Arithmetic (IEEE 754) occupies effectively fewer than 64, e.g., only 13 bits.

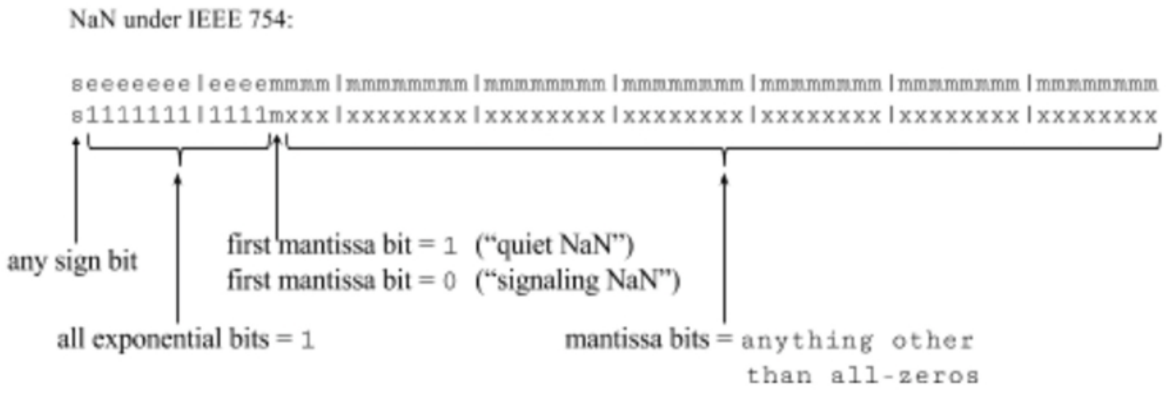

Fig. 1: NaN representation under IEEE754

Fig. 1 shows the representation of NaN under IEEE 754. The sign bit is allowed to be either zero or one, and the first mantissa bit is either zero (signifying a signaling NaN) or one (signifying a quiet NaN). The exponential bits must all be one. The remaining mantissa bits can be any value except the all-zero string.

Thus, a 51-bit "NaN space" comprising all mantissa bits except the first (leftmost) mantissa bit is available within a 64-bit NaN. Pointer tagging is a technique by which a 47-bit pointer is stored in the lower 51 bits of a register representing a 64-bit double type, with the remaining 6 bits used to store small amounts of information known as "tags". The small bits of tagged information are used, e.g., to reduce data movement between registers and RAM. To store and/or tag a pointer in a 64-bit NaN floating-point register requires code that performs operations such as bit-masking, bit-shifting etc., known as "boxing" the pointer. Interpreting a boxed pointer requires implementing corresponding "unboxing" code.

Some solutions store more than one 64-bit value into a single 64-bit register by carefully examining and determining circumstances when certain bits are redundant. For example, tagging approaches that cannot represent a large enough variety of data types can make the data types smaller to increase the number of tag bits available. If reduction of the size of data type is not possible, e.g., on 32-bit systems, more than one register can be used to store a variable. Although such approaches make some tag bits available, they increase register pressure, e.g., they result in the depletion of available registers, causing a spill of variables into RAM. A spill of register variables into RAM manifests as lower VM performance.

In some approaches, a subset of a large data type is stored in the heap in tagged structures that are larger than 64 bits, and the VM may represent these types by their 64 bit pointer addresses. By treating more data types as 64 bit pointers, fewer tag bits may be required, but the additional heap usage increases the garbage collection burden. Increased garbage collection manifests as lower VM performance. The smaller the subset of the 64-bit type that is allowed, the more likely a value is spilled into heap instead of running directly in a register. Often the balance that is struck in selecting the subset of a data type that goes to heap, and the value of such a subset is arbitrary and does not match well with the specification of the programming language that the VM runs. This can lead to a requirement that multiple steps be performed to check that a value obeys the semantics of the modeled language. This can result in reduced VM performance.

Thus, current solutions have drawbacks such as inability to encode enough types of 64-bit values, requirement of excessive boxing/unboxing steps, ability restricted to storing a smaller subset of a 64-bit value in the register, etc.

DESCRIPTION

To allow storage of variables within registers and reduce spillage into RAM, the present disclosure describes techniques of inverse pointer unboxing (or "inverse punboxing") that store doubles in pointer space, rather than in the heap. Canonical double space uses all 64 bits, whereas a pointer may occupy up to 47 bits, leaving no spare room.

Under inverse punboxing, the intended type of a register is determined by doing an unsigned integer interpretation and comparison. Thus, a value stored in the register is deemed to be a pointer if it is less than or equal to 2^{47}-1= 140,737,488,355,327. If a value stored in the register, when interpreted as an unsigned integer is greater than 2^{51}-1= 2,251,799,813,685,247, then it is deemed a bitwise-flipped double. This is inherently true for all bitwise-flipped non-NaN doubles, as well as the only NaN double produced, e.g., by the SSE2 floating point instructions.

This enables all values between 2^{47}-1 and 2^{51}-1, represented by bits 48 through 51 (a total of 3 bits) to be available to tag non-double non-pointer values. This leaves room to store 32-bit integers in the lowest 32 bits of a register by simply marking one of those three bits. For example, a three-bit pattern held within bits 48 through 51 can tag how the remaining bits should be interpreted. When one or more of these tag bits is set, the lowest 48 bits can be used to store booleans or indeed any other value of interest.

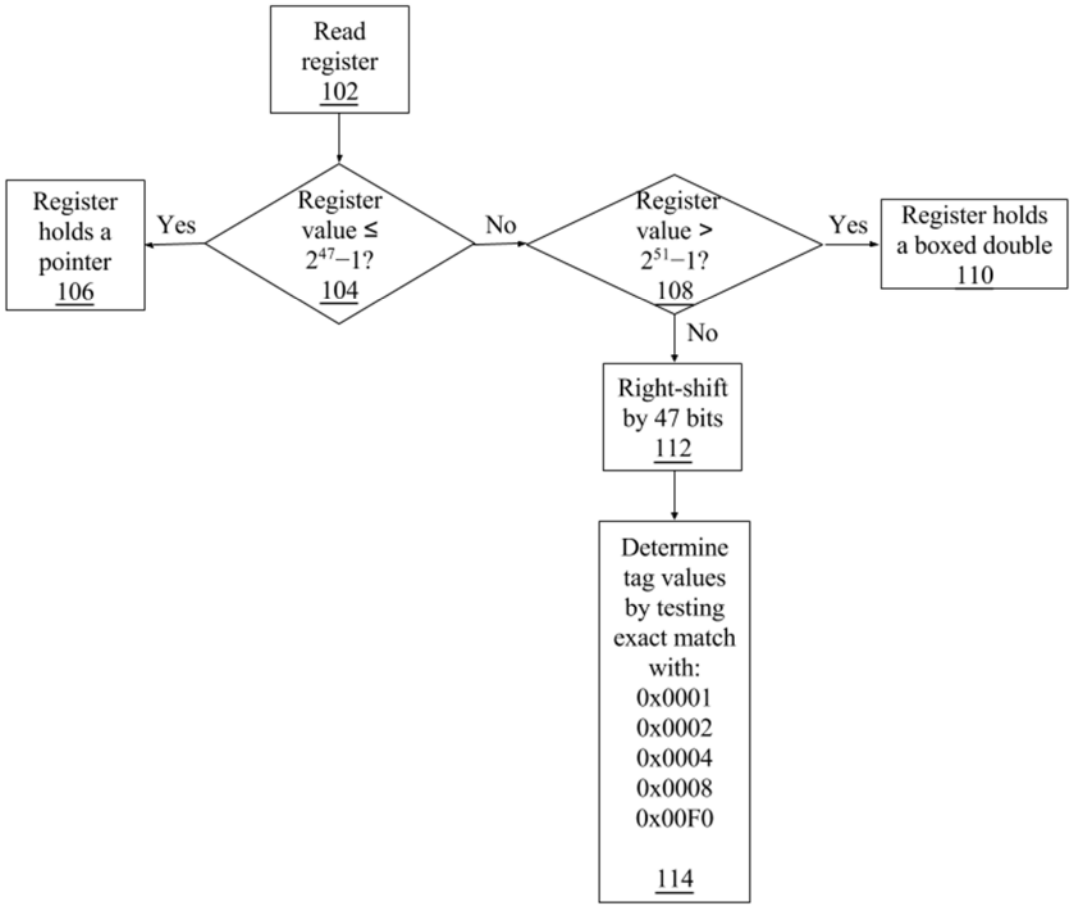

Fig. 2: Inverse punboxing

Fig. 2 illustrates inverse punboxing. A register is read (102) and its value compared, as an unsigned integer, to a threshold (104), e.g., 2^{47}-1. If the value of the register is less than or equal to the threshold, it is determined that the register holds a pointer (106). If the value of the register is greater than the threshold, a further comparison (108) is made with a second threshold, e.g., 2^{51}-1. If the register value, as an unsigned integer, exceeds the second threshold, then the register is deemed to hold a boxed double (110). If the register value is less than the second threshold, then the register is right-shifted (112) by an amount equal to the width of the pointer, e.g., 47 bits. The tag values stored in the bits between the first and second thresholds are determined by testing for an exact match (114) with bit patterns. For example, tag values held in bits 48-53 are tested by testing for exact match with the bit patterns *0x0001*, *0x0002*,

0x0004, *0x0008*, and *0x00F0*. The process of right shifting (112) and testing for exact match (114) confirms that the register value is greater than the maximum pointer value, and less than the minimum boxed double value, and that the appropriate tag bit is set. Alternative to testing for exact match with selected bit patterns, tag values (non-double, non-pointer values) can be distinguished by a numeric interpretation of the bits that hold them, e.g., bits 48-53. For example, rather than testing for exact match with *0x0001*, *0x0002*, *0x0004*, etc., a bit pattern *0x0003* can be used as well. This allows for more types of 32-bit (or 47-bit) data types to be stored in registers at the expense of being able to use those types in bitmaps. Although the foregoing description refers to techniques implemented in a 64-bit architecture, the techniques are applicable to other architectures, e.g., 32-bit architectures, etc.

Similar to doubles being boxed into pointer space, integers also can be thus boxed. Boxing an integer requires loading the proper 64-bit value and using OR to flip the mark bit. Other equivalent binary operations are possible, such as using add instructions, etc. Similar strategies apply to other taggable values, for example, Booleans can be stored in the lower 32 bits with the second mark bit set during the boxing process. Integers can be used via 32-bit registers (e.g., the eax, edi, esi, ebx, etc. registers in an x86 architecture) with no unboxing required. Doubles can be bitwise inverted to box or unbox. Fig. 3 shows an example of x86 assembly code that performs boxing and unboxing. Such code allows for fast virtual machines, even when boxing and unboxing are performed repeatedly.

```
; box and unbox
doubles pcmpeqd xmm0, xmm0 pandn xmmX, xmm0 ; box integer
mov rcx, 0x0001000000000000 or rax, rcx ; use a 64 bit boxed integer as a 32 bit integer
add eax, ecx ; typecheck pointer
mov rcx, 0x00007FFFFFFFFFFF cmp rax, rax jg not_pointer ; typecheck boxed double
mov rcx, 0x0007FFFFFFFFFFFF cmp rax, rax jl not_double ; typecheck boxed int
shr rax, 48 cmp rax, 0x0001 jnz not_int
```

Fig. 3: Example of x86 assembly code that can box and unbox

CONCLUSION

Techniques described herein enable speedy execution of virtual machines by enabling storage of variables of differing types within a single register. This disclosure provides techniques that store doubles in non-canonical pointer space. Per techniques of this disclosure, doubles and pointers can both be stored in the same register. Techniques disclosed herein reduce spillage of variables into RAM. The VM performance is based on a reduced requirement to access variables from RAM, since the techniques enable more variables to be stored in registers, which are faster to access than RAM.